Clases de instrucciones

VARIABLES ASIGNACION

simples: var = valor → ej: x =8 z= 4
compuestas : x=x+1 ; y=y-1 ; z =x+y;

Llamado a funciones : x= f(y) + g(6) -7

Swap (intercambia valores entre variables, no se usa en la practica): v1 :=: v2

Skip: termina inmediatamente y no tiene efecto sobre variables del programación.

ESTRUCTURAS DE CONTROL

Condicional :

- **Simple**: IF (bool) → instrucción simple o {} ELSE uso de llaves para mas de una instrucción.
  Otra manera de hacer lo mismo es: If(cond) S;
  If (cond) S1 else S2;
- **Alternativas Multiples**:

```
IF B1 → S1
□ B2 → S2
…..
□ Bn → Sn
FI
```

Esta estrucctura nueva, evalua primero todas las condiciones booleanas, y de las verdaderas toma una de manera no deterministica y la ejecuta. Si ninguna condicion es VERDADERA, el if no tiene efecto.

Iteracion :

- **Sentencias de Alternativa ITERATIVA multiple**: Las sentencias guardadas son evaluadas y ejecutadas hasta que todas las guardas sean falsas. La eleccion es no deterministica si mas de una guarda es verdadera

```
do B1 → S1
□ B2 → S2
…..
□ Bn → Sn
```

- **FOR-ALL**: forma general de repeticion e iteracion: El fuerpo de del FA se ejecuta 1 vez por cada combinacion de valores de las variables de iteración,. Si hay clausula SUCH-THAT (st), la variable de iteracion toma solo los calores para los que B es TRUE.
  Fa cuantificadores → secuenca de instrucciones Af
  Cuantificador ≡ variable := exp_inicial to exp_final st B
  ejemplo: fa i:= 1 to n, j:= i+1 to n st a[j] > a[i] :=: a[j] af

  EJ:
  - Inicializar un vector: fa 1:=1 to n → a[i] = 0 fa
  - transposicion de matriz: fa i:=1 to n, j:=i+1 to n → m[i,j] :=: m[j,i] af

- ordenacion de menor a mayt de un vector:
  fa i:=1 to n, j:=i+1 to n st a[i] > a[j] → a[i] :=: a[j] af
- **Otra opcion**:
  while (cond) S;
  for [i=1 to n, j=1 to n st (j mod 2=0)] Sentencia;

CONCURRENCIA: Todas las acciones es para generar concurrencia de procesos.

- **Sentencia CO**: Durante la ejecucion de CO, se crean Hilos de tareas para el trabajo en concurrente. Cuando se termina la ejecucion de estos hilos se retorna con el proceso original que los creo.

  Co s1 // ….. // sn oc

  Ejemplo **Cuantificadores**: crea n tareas concurrentes (hilos) en este caso se inicializan dos vectores en 0.

  co [i=1 to n] {a[i]=0 ; b[i] = 0} oc
- **Process**: otra forma de representar la concurrencia. Process se ejecuta en background.

  Process A {sentencias} → unico proceso independiente

  Cuantificadores: crea n tareas concurrentes iguales, que realizaran las mismas tareas. Salvo que cada proceso conocera a "i" con un valor diferente.

  Process B [i=1 to n] {sentencias} → n procesos independientes.

  Ejemplo:

  1. Se genera un proceso que imprime en pantalla los numero del 1 al 10.

     "process imprime10 {
     for [i=1 to 10] write(i);
     } "

  2. /Se generaran 10 procesos diferentes que imprimiran los numeros en funcion de que conoce cada proceso. Una vez creados los procesos se ejecutan todos juntos de manera no deterministica. Por lo tanto los numeros impresos pueden no estar ordenados del 1 al 10 como en el caso 1/

     "process imprime1 [i=1..10] {
     write (i);
     }"

ACCIONES ATOMICAS Y SINCRONIZACION:

- **ESTADO (contexto, para el context switch):** de un programa concurrente es el valor que tiene todas las variables compartidas y locales de cada proceso, el valor de los registros y demas en un determinado instante de tiempo.
- **Acción atomica**: cuando una sentencia se ejecuta, hay estados que son invisibles para otros procesos, los cuales evitan la interferencia durante la ejecucion de esta.
- **Interleaving** : Es el intercalado de acciones atomicas ejecutadas por procesos individuales.
- **Historia de un programa concurrente (TRACE)**: Una historia es una posible ejecucion real de un programa concurrente no deterministico, Por lo general la cantidad de historias que un programa puede tener es enorme, pero no todas son validas.
- **Interaccion**: Determina cuales historias son correctas.

Atomicidad de GRANO FINO:

Una sentencia de grano fino es aquella que durante la ejecucion de esta NO pueda ser interrumpida por otra sentencia.

Ejemplo de una sentencia NO atomica:

- A=B → esto no es dado que sucede en lenguaje maquina:
  - (i) Load posMemB, Reg
  - (ii) Store reg, posMemA
- X=X+X → esto no debido a que el valor de X puede ser modificado en cualquier momento.
  - (i) Load PosMemX. Acumulador
  - (ii) Add PosMemX. Acumulador
  - (iii) Store acumulador, PosMemX

Ejemplo de una sentencia SI atomica:

- y = 3 ; x=8 ; z= 9999 ej: (i) storage 3 , posMemY ;

NOTA: tener en cuenta que:

- Los sentencias atómicas son por ejemplo la asignación de un dato simple en una variable.
- Los valores se cargan y operan sobre registros. Y luego se almaenan los resultados en memoria.
- Cada proceso cuenta con su propio conjunto de registros (context swiching)
- Todo resultado intermedio de evaluar una expresión compleja se almacena en registros o en memoria privada del proceso.

Ejemplo 1: Cuáles son los posibles resultados con 3 procesadores. La lectura y escritura de las variables x, y, z son atómicas.

```
x = 0; y = 4; z=2;
co
  x = y + z      (1)
  // y = 3       (2)
  // z = 4       (3)
oc
```

(1) Puede descomponerse por ejemplo en:
(1.1) Load PosMemY, Acumulador
(1.2) Add PosMemZ, Acumulador
(1.3) Store Acumulador, PosMemX
(2) Se transforma en: Store 3, PosMemY
(3) Se transforma en: Store 4, PosMemZ

- y = 3, z = 4 en todos los casos.
- x puede ser:
  - 6 si ejecuta (1)(2)(3) o (1)(3)(2)
  - 5 si ejecuta (2)(1)(3)
  - 8 si ejecuta (3)(1)(2)
  - 7 si ejecuta (2)(3)(1) o (3)(2)(1)
  - 6 si ejecuta (1.1)(2)(1.2)(1.3)(3)
  - 8 si ejecuta (1.1)(3)(1.2)(1.3)(2)
  - .....

Ejemplo 2: Cuáles son los posibles resultados con 2 procesadores. La lectura y escritura de las variables x, y, z son atómicas.

```
x = 2; y = 2;
co
  z = x + y          (1)
  // x = 3; y = 4;   (2)
oc
```

(1) Puede descomponerse por ejemplo en:
- (1.1) Load PosMemX, Acumulador
- (1.2) Add PosMemY, Acumulador
- (1.3) Store Acumulador, PosMemZ

(2) Se transforma en:
- (2.1) Store 3, PosMemX
- (2.2) Store 4, PosMemY

x = 3, y = 4 en todos los casos.
z puede ser: 4, 5, 6 o 7.

Nunca podría parar el programa y ver un estado en que x+y = 6, a pesar de que z = x + y si puede tomar ese valor

Atomicidad de grano fino

Ejemplo 3: "Interleaving extremo" (Ben-Ari & Burns)
Dos procesos que realizan (cada uno) *N* iteraciones de la sentencia *X=X+1*.

```
int X = 0
Process P1
{ int i
  for [i=1 to N] → X=X+1
}
Process P2
{ int i
  fa [i=1 to N] → X=X+1
}
```

Cuál puede ser el valor final de *X*?
- *2N*
- entre *N+1* y *2N-1*
- *N*
- < *N* (incluso 2…)

¿Cuando valdrá *2N*?
En cada iteración ….
1. Proceso 1: *Load X*
2. Proceso 1: *Incrementa su copia*
3. Proceso 1: *Store X*
4. Proceso 2: *Load X*
5. Proceso 2: *Incrementa su copia*
6. Proceso 2: *Store X*

¿Cuando valdrá *N*?
En cada iteración ….
1. Proceso 1: *Load X*
2. Proceso 2: *Load X*
3. Proceso 1: *Incrementa su copia*
4. Proceso 2: *Incrementa su copia*
5. Proceso 1: *Store X*
6. Proceso 2: *Store X*

Lo importante para los programas concurrentes es tener la habilidad de reconocer las situaciones criticas para darle una sincronizacion optima para que los diferentes INTERLEAVING (como se desarrolla el paso a paso de la aplicación) nos de por resultado el que nosotros esperamos. Aquellas situaciones concurrentes pero que no son criticas pueden no ser manejadas con precisión.

Referencia Crítica: es una expresion que se refiere a una variable que esta siendo modificada por otro proceso. SE ASUME QUE UNA REF CRITICA ES A UNA VARIABLE SIMPLE LEIDA Y ESCRITA ATÓMICAMENTE.

Propiedad a Lo sumo una vez ASV

Una sentencia de asignación **x = e** satisface la propiedad de ***"A lo sumo una vez"*** si:

1) **e** contiene a lo sumo una referencia crítica y **x** no es referenciada por otro proceso, o
2) **e** no contiene referencias críticas, en cuyo caso **x** puede ser leída por otro proceso.

Una expresiones ***e*** que no está en una sentencia de asignación satisface la propiedad de ***"A lo sumo una vez"*** si no contiene más de una referencia crítica.

Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez

Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución ***parece*** atómica, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos:

- `int x=0, y=0;`
  `co x=x+1 // y=y+1 oc;`
  No hay ref. críticas en ningún proceso.
  En todas las historias ***x = 1*** e ***y = 1***

- `int x = 0, y = 0;`
  `co x=y+1 // y=y+1 oc;`
  El 1er proceso tiene 1 ref. crítica. El 2do ninguna.
  Siempre ***y = 1*** y ***x = 1 o 2***

- `int x = 0, y = 0;`
  `co x=y+1 // y=x+1 oc;`
  Ninguna asignación satisface ASV.
  Posibles resultados: ***x= 1 e y =2*** / ***x = 2*** e ***y = 1***
  Nunca debería ocurrir x = 1 e y = 1 → ERROR

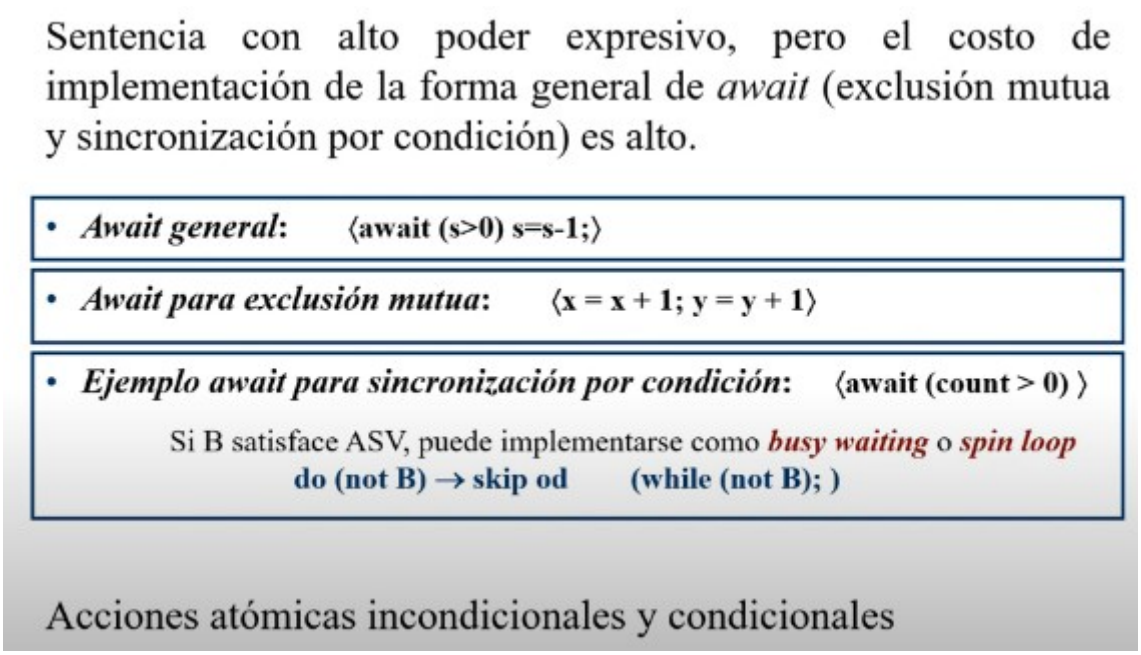
Sincronizacion por exclusion mutua Y por condicion <AWAIT (boolean) S;> : Si una expresión no cumple con la propiedad ASV, entonces se debe usar de esta manera. Buscando que ningún proceso pueda interferir en la ejecución de otro, asegurando la atomicidad de la ejecución y la confiabilidad de los datos.

TIPOS:

S = sentencias: B = booleano (TRUE ingresa)

- < S > : estas son sentencias atómicas incondicionales que se garantiza que se termina, a esto se lo denomina sincronización por exclusión mutua.

- <Await (B) S;> : esta sincronizacion por exclusion mutua <S> y sincronizacion por condicion <Await (B)>



Acciones atomicas Incondicionales: Son las Await que no usan un booleano, Se hara si o si.,
Acciones atomicas Condicionales: Son las Await que usan booleanos donde la ejecucion depende de esto.

Ejemplo: productor/consumidor con buffer de tamaño N.

```
cant: int = 0;
Buffer: cola;

process Productor
 { while (true)
      Generar Elemento
      <await  (cant < N); push(buffer, elemento); cant++ >
 }

process Consumidor
 { while (true)
      <await  (cant > 0); pop(buffer, elemento); cant-- >
      Consumir Elemento
 }
```

- Elemento es una variable propia de cada proceso (no comun)

PARTE 3

Propiedad: es un atrihuto verdadera en cualquiera de las historias dde ejecicion del mismo. Y se dividen en:

- **propiedad de seguridad (Safety) :**"Nada malo le ocurre a un proceso, asegurando los estados consistentes. Una falla de seguridad indica que algo anda mal.
  Ejemplos de esta propiedad : Exclusion mutua, ausencia de interferecia entre procesos.
- **Propiedad de vida (liveness) :** esta marca que no existan DEADLOCK, haciendo que el programa progrese correctamente. Una falla de vida indica que las cosas dejan de ejecutar.
  Ejemplos de esta: **Terminacion**(asegurar que un pedido de servicio sera atendido, que un mensaje llegue a destino , que un proceso llegue a su seccion critica, etc. → Esto dependera de las politicas de scheduling de la pc. Ausencia de inanicion!!!

Correccion total : nos asegura que el programa concurrente siempre termine y que este Sea correcto.

El FAIRNESS (JUSTICIA?) : trata de garanticar que los procesos avancen sin importar que hacen los demas.

Una accion atomica en un proceso es **ELEGIBLE** si es la prox accion en el proceso que sera ejecutada. Si hay varios procesos, hay varias acciones elegibles (la politica del Scheduling define cual seria esta).

LAS POLITICAS DE SCHEDULING SE Clasifican en:

- **Fairness incondicional**: esta es fair (justo, equitativo) si toda accion atomica incondicional (sentencias que cumplen con ASV o AWAIT sin condicion <S>) que es elegible, eventualmente es ejecutada.
  Ej round-robin, ejecucion paralela (1 proceso por cada procesador)

- **Fairness debil**: esto sucede si:
  - es incondicionamente fair y ademas cumple con :
    - toda accion atomica condicional que se vuelve elegible en un proceso en algun momento va ser ejecutada si la condicion es verdadera y permanece de esta manera hasta que es vista por el proceso que ejecuta la accion atomica condicional (sin cambiar a falso). Recordemos que una condicion puede ser verdadera por un tiempo pero si cambia antes de poder evaluar la condicion, es como si nunca hubiese existido. Ej: round-robin
- **Fairness Fuerte**: Una política de **scheduling es fuertemente fair** si:
  - (1)Es incondicionalmente fair y ademas:
    - (2)Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.
      Son procesos que continuamente varian el estado de una condicion True/false.

Ejemplo: ¿Este programa termina?

```
bool continue = true, try = false;
co while (continue) { try = true; try = false; }
 // ⟨await (try) continue = false⟩
oc
```

No es simple tener una política que sea práctica y fuertemente fair. En el ejemplo anterior, con 1 procesador, una política que alterna las acciones de los procesos sería fuertemente fair, pero es impráctica. Round-robin es práctica pero no es fuertemente fair.